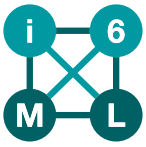


Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.

Learning General Policies for Qualitative Numeric Planning Problems

Bachelor Thesis

Presented by

Marleen König
446031

Supervised by Dr. Till Hofmann

1st Examiner Prof. Hector Geffner, Ph.D.

2nd Examiner Prof. Gerhard Lakemeyer, Ph.D.

Aachen, November 17, 2025

Abstract

Generalized planning provides a powerful approach to address entire classes of planning problems, instead of having to plan each instance individually. Qualitative Numeric Planning (QNP) expands Classical Planning by introducing variables with non-negative real values and qualitative effects. To make classical-planning methods applicable to QNP problems and support generalization across instances, we adapt existing formulations and solvers. We extend the existing definition for QNPs with arbitrary numeric comparisons between variables and constants, as well as assignment effects, and introduce a lifted representation. To keep the state space finite, we construct comparison graphs that group states similar to original QNPs. We define a feature space that handles functions as well as predicates, select the most significant features, and then learn a general policy that solves the planning class using a Min-cost SAT formulation with a fairness-based termination check for loops. The experiments show that the proposed methods are capable of solving generalized versions of existing QNP problems, as well as continuous-space variants of classical problems such as Delivery.

Contents

1 Introduction

Generalized planning aims to find solutions that are not tied to a single problem instance but instead solve a whole class of instances that share a common domain description while differing in their initial states, objects, and goal conditions. One approach is to compute general policies, which map abstract state descriptions to actions rather than concrete action sequences. Generalized planning eliminates the need to replan from scratch for every new instance and therefore promises considerable advantages in scalability and reusability.

Classical planning represents states by finite sets of propositional atoms, and actions are characterized by the preconditions that must hold and the effects they produce on those states. Many practical problems, however, require modeling resources, distances, or other continuous quantities that are not conveniently captured with pure propositional symbols. Qualitative Numeric Planning (QNP) extends the classical model by introducing numerical variables that range over non-negative real values and by augmenting actions with qualitative increase ($\uparrow$ x) and decrease ($\downarrow$ x) effects. However, there is currently no method for generating general policies for QNPs. For example, the continuous delivery problem describes a number of packages, a gripper, and a target, each with coordinates represented as real values. The gripper can move in any direction and pick up and put down a package when they are at the same location. The goal is to deliver all packages to the target position. We can compute a plan for a single instance, but even a slight change, such as adding another package, requires rebuilding the plan from scratch for the new problem. A general policy describes rules that are able to solve every instance of this domain.

This work realizes that idea by proposing a lifted representation to handle structurally similar variables. For example, two variables describing the x-coordinates of different objects, x_a and x_b, are unified into $x(o)$ with $o \in \{a, b\}$. Additionally, we extend the semantics of QNPs, such that we permit comparisons of numerical variables n, m to arbitrary constants c and to each other ($n < c, n = c, c < n$ and $n < m, n = m, m < n$), and introduce assignment effects $assign(n, c)$ and $assign(n, m)$ alongside qualitative increments and decrements.
To learn a general policy, we adapt the Min-Cost SAT approach of Hofmann et al. 2024 [6], enriching its Answer Set Program (ASP) with adaptations needed to handle the specifics of the extended QNPs, and a fairness-based termination check to prevent non-terminating loops caused by infinite alternation of opposing effects. To obtain a finite state space needed for the SAT approach, we introduce comparison graphs. These are compact, non-deterministic transition and state-space representations, whose nodes aggregate ground states that agree

on all relevant comparisons and whose edges apply bounded changes, preventing over- or undershooting numeric thresholds.
The proposed pipeline solves the SAT encoding by identifying features that can distinguish states and transitions. These features are built automatically over the atoms and functions of the states. We extended the existing grammar [4] which describes concepts and roles as abstractions for predicates, with numeric frames defined over grounded functions. This enables new, complex numerical features, such as the maximal distance of any robot to the goal target. Finally, we explain the policy extraction and evaluate the approach on four domains (Snow, GoTo, Sailing, and Delivery). We are able to show that the resulting policies solve all benchmark instances, most notably Delivery with continuous space in one and two dimensions.

2 Background

This chapter summarizes for classical, generalized, and qualitative numerical planning. It also introduces the feature pool grammar and gives a definition of fairness assumptions in Fully Observable Non-Deterministic (FOND) planning.

2.1 Classical planning

Planning is the process of formulating a sequence of actions or steps to achieve specific goals or objectives. A classical planning problem [2] is defined as a tuple $P = \langle F, I, G, O \rangle$, where

- F is a set of atoms.
- I is a set of initially given atoms over V.
- G is a set of goal literals.
- O is a finite set of actions over F. Each action is defined by its preconditions and effects $\langle \text{pre}(a), \text{eff}(a) \rangle$ each consisting of a set of atoms.

Given a state s, an action a is **applicable** in s if all atoms in pre(a) are satisfied in s. Let $A(s)$ denote the set of all actions applicable in a state s. The result of applying an action to a state is described by the successor state function $f(a, s) = s'$, where the successor state is defined by incorporating the changes specified in eff(a). A **solution** to the planning problem is a finite action sequence $a_0, ..., a_n$ such that, if successively applied to a initial state s_0, the resulting state s_n satisfies the goal literals in G. A **policy** π_p is a function which maps states to actions, this induces a solution if applied to a planning problem P.

2.2 Qualitative Numerical Planning

A QNP is an abstraction of quantitative planning problems that considers variables with non-negative real values [10]. Formally, a QNP is defined as a tuple $Q = \langle X, I, G, O \rangle$, where

- X is a set of variables. Each variable $x \in X$ may be numerical or boolean. If x is a numerical variable, we consider the set of literals $x = 0$ and $x \neq 0$.
- I is the set of initially true literals from X.
- G is the set of goal literals.
- O is the set of qualitative action operators. Each action $a \in O$ is defined by the set of preconditions and effects $\langle \text{pre}(a), \text{eff}(a) \rangle$. For a quantitative variables x the preconditions specify either $x = 0$ or $x \neq 0$, and the effects can increase ($\uparrow x$) or decrease ($\downarrow x$) the value.

A **state** $s \in S$ represents a (partial) assignment of the variables in X. The set of initial states S_0 includes all states that satisfy the literals in I. An action is **applicable** in a state s if all of its preconditions are satisfied in s. The set of all applicable actions in state s is denoted by $A(s)$. Applying an action $a \in A$ to s is described by the **successor state function** $F(a, s)$, resulting in a state s' where the effects of the action are applied:

- For boolean variables p, if p in eff(a) then $p \in s'$ and if $\neg p$ in eff(a) then $p \notin s'$.
- For numeric variables n, its value strictly increases (decreases, but cannot become negative), if inc(n) $\in$ eff(a) (dec(n)).

If a variable is not in the effects, it remains unchanged.
A **solution** is a sequence of applicable actions that transforms some initial state $s_0 \in S_0$ into a state satisfying all the goal conditions in G. A **trajectory** $s_0, a_0, s_1, a_1, s_2, ...$ is a sequence of states and actions, starting with a initial state s_0 and for every step i it holds that $a_{i+1} \in A(s_i)$ and $s_{i+1} \in f(a_i, s_i)$. A **transition** (s_i, s_{i+1}) with a_i is ϵ-**bounded**, if each numerical change is larger or equal to ϵ, unless a decrease would make a variable negative, in which case it is only reduced to 0. An trajectory is ϵ-bounded, if all transitions are ϵ-bounded.
A **policy** π_q is a partial function mapping states s of a problem instance Q into sets $\pi(s)$ of actions of Q. If π_q yields a finite, ϵ-bounded plan for every initial state that terminates in a goal state, it is said to solve a problem. π_q is said to terminate, if all instantiated, ϵ-bounded trajectories are finite. To check for termination of a policy, Srivastava et al. [10] constructed the **Sieve Algorithm**. The algorithm works by incrementally identifying and pruning transitions that reduce a variable which is not increased by another transition. If the remaining subgraph without removable edges is cyclic the policy is identified as non-terminating, terminating otherwise.

Example Snow

To illustrate the concepts behind QNPs and convey the ideas behind our approach, we introduce the snow domain and treat it as the running example for the approach in chapter 4.
Formally, Snow is defined as:

$$
\begin{aligned}
Q_1 &= \langle X, I, G, O \rangle \\
&X = \{atDW, sw, sd\} \\
&I = \{\neg atDW, sw \geq 0, sd \geq 0\} \\
&G = \{sw = 0, sd = 0\} \\
&O = \{shovel()\langle sw > 0; dec(sw)\rangle, \\
&moveSBtoDW()\langle sw = 0; atDW\rangle, \\
&blow()\langle atDW; dec(sd), inc(sw)\rangle\}
\end{aligned}
$$

In this domain, the variables are two numerical values indicating the amount of snow in the driveway and the walkway $sw \equiv$ |snow on walkway|, $sd \equiv$ |snow on driveway|, and a boolean describing whether the snowblower is in the driveway $atDW \equiv at(driveway)$. The actions include *shovel*() which decreases *sw*, *moveSBtoDW*() to move the snowblower to the driveway with the precondition $sw = 0$, and *blow*() with the precondition that atDW is true and then decreases *sd* but also increases *sw*. The goal is to clear all snow from both locations.

The solution policy π_{Q_1} of Snow is the following:

$$sw > 0 \mapsto\downarrow sw$$
$$sw = 0, \neg atDW \mapsto atDW$$
$$atDW, sd > 0 \mapsto\downarrow sd$$

2.3 DLPlan

DLPlan [4] is a library that uses description logic to extract features from the states of classical planning problems. The inputs are a set of predicates, objects, and states, and incrementally generates a set of numerical and boolean features. These features provide an abstraction of the planning problem; they allow us to describe and distinguish states and transitions, and to generalize by grouping similar states and transitions. These features can then be used to learn a general policy (section 2.5).
The library constructs, for every k-ary predicate p with $0 \leq i, j < k$ and the universe Δ^s (the the set of objects occurring in state s), the following primitive concepts (unary) and roles (binary):

- $(p[i])^s = \{\, c_i \in \Delta^s \mid p(c_0, \ldots, c_i, \ldots, c_{k-1}) \in s \,\}$,
- $(p[i,j])^s = \{\, (c_i, c_j) \in \Delta^s \times \Delta^s \mid p(c_0, \ldots, c_i, \ldots, c_j, \ldots, c_{k-1}) \in s \,\}$.

We continue by iteratively constructing more complex concepts and roles, given C, D are concepts and R, S roles.
Compositional concepts:

- universal concept $\top$ where $\top^s = \Delta^s$,
- bottom concept $\bot$ where $\bot^s = \emptyset$,
- intersection $\mathbf{C} \sqcap \mathbf{D}$ where $(C \sqcap D)^s = C^s \cap D^s$,
- union $\mathbf{C} \sqcup \mathbf{D}$ where $(C \sqcup D)^s = C^s \cup D^s$,
- negation $(\neg \mathbf{C})$ where $(\neg C)^s = \Delta^s \setminus C^s$,
- difference $(\mathbf{C} \setminus \mathbf{D})$ where $(C \setminus D)^s = C^s \setminus D^s$,
- existential restriction $\exists \mathbf{R}.\mathbf{C}$ where $(\exists R.C)^s = \{\, a \mid \exists b : (a,b) \in R^s \wedge b \in C^s \,\}$,
- universal restriction $\forall \mathbf{R}.\mathbf{C}$ where $(\forall R.C)^s = \{\, a \mid \forall b : (a,b) \in R^s \rightarrow b \in C^s \,\}$,
- constant concept $\mathbf{c}$, one for each domain constant c, where $c^s = \{c\}$.

Compositional roles:

- universal role $\top$ where $\top^s = \Delta^s \times \Delta^s$,
- intersection $\mathbf{R} \sqcap \mathbf{S}$ where $(R \sqcap S)^s = R^s \cap S^s$,
- union $\mathbf{R} \sqcup \mathbf{S}$ where $(R \sqcup S)^s = R^s \cup S^s$,
- negation $\neg\mathbf{R}$ where $(\neg R)^s = \top^s \setminus R^s$,
- inverse $\mathbf{R^{-1}}$ where $\left(R^{-1}\right)^s = \{\, (b, a) \mid (a, b) \in R^s \,\}$,
- composition $\mathbf{R} \circ \mathbf{S}$ where $(R \circ S)^s = \{\, (a, c) \mid (a, b) \in R^s \wedge (b, c) \in S^s \,\}$,
- transitive closure $\mathbf{R^+}$ where $(R^+)^s = \bigcup_{n \geq 1} (R^s)^n$,
- transitive reflexive closure $\mathbf{R^*}$ where $(R^*)^s = \bigcup_{n \geq 0} (R^s)^n$,
- restriction $\mathbf{R} \mid \mathbf{C}$ where $(R \mid C)^s = R^s \cap (\Delta^s \times C^s)$,
- identify $\mathrm{id}(\mathbf{C})$ where $(\mathrm{id}(C))^s = \{\, (a, a) \mid a \in C^s \,\}$.
- iterated composition $(\mathbf{R^s})^{\mathbf{n}}$ is constructed inductively with $(R^s)^0 = \{\, (a, a) \mid a \in \Delta^s \,\}$ and $(R^s)^{n+1} = (R^s)^n \circ R^s$

From these concepts C, D and roles R, S, T, we can create boolean and numerical freatures. The boolean are defined as follows:

- empty feature $\mathrm{Empty}(\mathbf{C})$ where $(\mathrm{Empty}(C))^s = \top$ iff $C^s = \emptyset$,
- concept inclusion $\mathbf{C} \sqsubseteq \mathbf{D}$ where $(C \sqsubseteq D)^s = \top$ iff $C^s \subseteq D^s$,
- role inclusion $\mathbf{R} \sqsubseteq \mathbf{S}$ where $(R \sqsubseteq S)^s = \top$ iff $R^s \subseteq S^s$,
- nullary $\mathrm{Nullary}(\mathbf{p})$ where $(\mathrm{Nullary}(p))^s = \top$ iff p is a nullary state predicate and $p \in s$.

The numerical features are defined as follows:

- count $\mathrm{Count}(\mathbf{C})$ where $(\mathrm{Count}(C))^s = |C^s|$,
- concept distance $\mathrm{dist}(\mathbf{C}, \mathbf{R}, \mathbf{D})$ where $(\mathrm{dist}(C, R, D))^s$ is the smallest $n \in \mathbb{N}_0$ such that there are objects $o_0, \ldots, o_n$ with $o_0 \in C^s$, $o_n \in D^s$, and $(o_i, o_{i+1}) \in R^s$ for all $0 \leq i < n$. If C^s is empty or no such n exists, then $(\mathrm{dist}(C, R, D))^s = \infty$,
- the sum concept distance $\mathrm{sdist}(\mathbf{C}, \mathbf{R}, \mathbf{D})$ where $(\mathrm{sdist}(C, R, D))^s = \sum_{x \in C^s} \mathrm{dist}^s(\{x\}, R, D)$,
- role distance $\mathrm{rdist}(\mathbf{R}, \mathbf{S}, \mathbf{T})$ where $(\mathrm{rdist}(R, S, T))^s$ is the smallest $n \in \mathbb{N}_0$ such that there are objects $a, o_0, \ldots, o_n$ with $(a, o_0) \in R^s$, $(a, o_n) \in T^s$, and $(o_i, o_{i+1}) \in S^s$ for all $0 \leq i < n$. If R^s is empty or no such n exists, then $(\mathrm{rdist}(R, S, T))^s = \infty$,
- the sum role distance $\mathrm{srdist}(\mathbf{R}, \mathbf{S}, \mathbf{T})$ where $(\mathrm{srdist}(R, S, T))^s = \sum_{r \in R^s} \mathrm{rdist}^s(\{r\}, S, T)$.

In this thesis, we extend DLPlan to also accept (grounded) functions as input and to generate additional features over them.

2.4 FOND with explicit fairness assumptions

FOND extends classical planning with non-deterministic actions that may have multiple possible outcomes. Rodriguez et al. [8] formalized explicit fairness assumptions for FOND planning, and denoted this setting Fully Observable Non-Deterministic Planning with Explicit Fairness Assumptions.

Formally, the fairness assumptions of a problem P are defined as pairs $A_i/B_i, i = 0, \dots, n$, where A_i is a set of non-deterministic actions, and B_i is a set of actions in P which is disjoint from A_i. A_i/B_i said to be **fair**, if any state trajectories that contains infinite occurrences of $a \in A_i$ in a state s and finite occurrences of all $b \in B_i$, must also contain infinite occurrences of all possible outcomes of a in s.
The fairness assumptions by Rodriguez et al. [8] represent a unifying mechanism for handling non-determinism in actions. In section 4.3.2 we will reuse this principle to handle non-determinism by formulating the termination conditions for solutions to generalized QNPs.

2.5 Generalized planning

Generalized planning [2] addresses a class of classical planning problems that share the same actions, domains, and certain structural similarities, but may differ in initial state, objects and goal conditions.
For these similar Problems we adopt a lifted representation [5], where a classical planning problem is defined by a pair $P = \langle D, I \rangle$, with D denoting the domain containing a set of action schemas and $I = \langle O, s_0, G \rangle$ representing the instance. Here, O is a set of objects, s_0 is the initial state, and G contains the goal condition. This language is lifted in a sense that the effects and preconditions of the actions are expressed with atoms $p(x_1, ..., x_k)$, where p is a predicate symbol of arity k, and each term x_i is an argument. Ground atoms in the states and grounded actions result by instantiating the arguments x_i with objects $o_i \in O$.
Furthermore, a **generalized classical planning problem** is a set $\mathfrak{G}$ of classical planning instances that all share the same domain D, and have different instances I. A solution to a generalized planning problem is given by a **general policy** π. This policy consists of a set of rules $C \mapsto E$, where C is a set of feature conditions and E describes the effects in a set of feature value changes. The rules hold the same grammar as QNPs, where features can be boolean p or numerical n, conditions may be $p, \neg p, n = 0, n \neq 0$, the effects can include $p, \neg p, n \uparrow, n \downarrow$, and features left unmentioned by the rule remain unchanged.
For a specific instance $P \in \mathfrak{G}$, the general policy induces a **concrete policy** π_P. Applied to a given state s of P, π_P maps to transitions (s, s') according to a rule R. Concretely, the conditions in C must be satisfied by the current state s, and the feature values must be updated according to the effects defined in E.

3 Related Work

QNPs have been proposed as an abstraction in classical planning [9] in order to discover general policies for a class of planning problems. In this work, we build on QNP abstractions to solve an entire class of QNPs rather than a single instance. This section first presents an established method for solving QNPs and then provides a brief overview of multiple strategies in generalized planning.

3.1 Solving Qualitative Numerical Planning

Bonet et al. [3] describe a sound and complete reduction of QNPs to Fully Observable Non-Deterministic (FOND) problems. This implies that any QNP problem can be converted into a FOND problem, enabling the use of a FOND planner to solve it. However, this method does not easily extend to a class of qualitative numerical planning problems, as it is designed to handle individual instances only.

3.2 Learning generalized policies

Various methods for learning a general policy have been proposed in the classical setting. They aim to derive single compact solutions that can solve multiple instances within the same domain. Below is a brief overview of various other approaches:

- Illanes et al. 2019 defined a quantified planning model in which initial states and goals can include existential and universal quantifiers to abstract from individual objects. At solve time, the LOOM algorithm synthesizes a partial policy that maps partial quantified states to partially grounded actions, and at execution time on any concrete instance, the system binds the abstract parameters to concrete objects of the required type whose preconditions hold in the current state. This collapses many symmetric ground plans into one compact, reusable policy that can be instantiated quickly without grounding all actions over all objects. [7]
- Ståhlberg et al. 2022 proposed learning general policies directly from small, lifted STRIPS instances using a simple Graph Neural Network (GNN). The GNN maps any state s to a scalar value V(s), and the greedy policy selects actions that lead to successor states with minimal V. The learned policies generalize to larger instances and often yield near-optimal plans without supervision or heavy feature engineering. When generalization

fails due to expressiveness or long-distance reasoning limits, adding lightweight “derived atoms” (role compositions or transitive closures) restores coverage. [11]

- Aichmüller et al. 2025 shows how to discover common subgoal structures (“sketches”) that split planning problems into smaller subproblems that can be solved efficiently. They cast sketch learning as a deep reinforcement learning task by learning policies over states reachable via IW(k), a width-based search algorithm. Although the decompositions are neural rather than rule-based, they often align with clear subgoal patterns and generalize from small to larger instances. [1]

4 Approach

In order to apply generalized planning to QNPs, we explain the following proposed methods in this chapter:

1. We define a variant of QNPs that allows comparisons with arbitrary constants and other numeric variables. The effects are redefined, such that we allow not only for lower-bounded increases and decreases, but also provide an upper-bound which prevents overshooting. Additionally, a new effect provide the possibility to assign a variable to a new value or to the value of another function.
2. To abstract further, we define a lifted representation, similar to lifted STRIPS.
3. The state space of the extended QNPs is represented by comparison graphs, which continue to allow us to work within a finite space while handeling the extended QNPs.
4. Building on that foundation, we define a suitable feature space for numerical functions, which extends the existing feature space of the [4] library.
5. We build a SAT solver inspired by [6] with additional fairness assumptions as termination conditions [8] to find the features and relevant transitions.
6. From the solution of the solver, we are then able to extract a general policy, which solves an entire class of (extended) QNP problems.

4.1 Extending Qualitative Numerical Planning

Srivastava et al [10] introduced QNPs to incorporate problems involving non-negative real variables. However, in their formulation, numerical variables are limited to two comparisons. For a numerical variable n, we can have the conditions $n = 0$ and $n > 0$. Actions can then increase or decrease these variables by an arbitrary amount. This definition is limited and does not suffice for more complex domains such as delivery, where it is crutial to be able to express whether or not the coordinates of a robot and the target are the same. For example, a state describes the one dimensional space, where the target is located at 10.0, the gripper at 10.0 and the package that needs to be delivered is at 5.5. In traditional QNPs the abstraction could only describe $0 < x_t, 0 < x_g, 0 < x_p$ none of which describe the relevant information, which would be $x_p < x_t = x_g$.
Therefore, to enhance expressiveness, we extend this representation by allowing numerical variables to be compared against any real constant or against any other numeric variable in the problem. This means that for any variables n, m, and constant $c \in \mathbb{R}$, conditions can now include:

- $n < c$, $n = c$, $n \neq c$, and $n > c$.
- $n < m$, $n = m$, $n \neq m$, and $n > m$,

Furthermore, we allow actions to additionally assign a numerical variable to the value of another numerical variable or to a constant. Actions can now include effects such as:

- $assign(n, m)$, $assign(n, c)$,
- $inc(n)$, $dec(n)$.

States of the extended QNPs are thus defined by the set of all predicates that are true and the set of equalities or inequalities of the numerical variables and values. In the initial state, as per the closed world assumption, all predicates not mentioned are assumed to be false and all numerical variables have to be equal to some value, otherwise they are assumed to initially equal 0. The value of a numerical variable n in a state is denoted by n^s.
In QNPs increases and decreases are lower-bounded by $\epsilon - bounded$ transitions [10]. For our extended QNPs we have to define an upper bound as well. Without these constraints, there is no safeguard against over- or undershooting target values, potentially making instances unsolvable. With a state s, an applicable action $a \in A(s)$ and the successor $s' = F(a, s)$, we define ω - **bounded transitions** (s, a, s') as such that we increase or decrease a variable only until the next change in comparison. Formally, a transition (s, a, s') is called $\omega - bounded$, if given N as the set of numerical variables of the problem:

1. For every numerical variable n with $\text{inc}(n) \in \text{eff}(a)$, there is a $\delta \in [0, \omega]$ such that $n^{s'} = n^s + \delta$, where $\omega = \min\{\, n^s - m^s \mid m \in N \setminus \{n\},\ m^s < n^s \,\}$
2. For every numerival variable n with $\text{dec}(n) \in \text{eff}(a)$, there is a $\delta \in [0, \omega]$ such that $n^{s'} = n^s - \delta$, where $\omega = \min\{\, n^s - m^s \mid m \in N \setminus \{n\},\ m^s < n^s \,\}$.

In the following, all transitions are assumed to be $\omega - bounded$ and $\epsilon - bounded$.
This extended QNP is still only an abstraction of quantitative numeric problems, however it is more expressive than QNPs and covers a wider range of problems, including continuous delivery.

4.1.1 Lifted Representation

In traditional QNPs variables are ground, meaning they don't have functions with parameters and only describe variables without referring to objects. To allow generalizing over varying numbers of objects in QNPs, we need to introduce a lifted representation. We are subsequently defining the problems as a class of problems over which we can generalize.
We define a lifted QNP as a pair $P = \langle D, I \rangle$, with D denoting the domain and I representing the instance. Similar to the classical setting, the domain D contains a set of action schemas and the instance $I = \langle O, s_0, G \rangle$ consists of a set of objects O, an initial state s_0, and the goal condition G. This lifted language allows us to express the preconditions and effects of actions using atoms

$p(x_1, ..., x_l)$ and $f(x_1, ..., x_k)$, where p is a predicate symbol of arity l and f a function of arity k respectively, and each term x_i is an argument. Ground atoms in the states, the goal conditions and grounded actions result by instantiating the arguments x_i with objects $o_i \in O$.
The notation $v = f^s(o)$ describes the value v that the function f given the objext o is equal to in the state s.
To visualize the new representation, the Snow domain and one problem instance is defined below.

Listing 4.1: Lifted QNP Domain for Snow

```
(define (domain snow)
  (:types walkway driveway - location)
  (:predicates (atDW))
  (:functions (snow ?l - location))
  (:action shovel
    :parameters (?w - walkway)
    :precondition (> (snow ?w) 0)
    :effect (decrease (snow ?w)))
  (:action move-snowblower
    :parameters (?w - walkway ?d - driveway)
    :precondition (and (= (snow ?w) 0) (not (atDW)))
    :effect (atDW))
  (:action snow-blower
    :parameters (?w - walkway ?d - driveway)
    :precondition (and (atDW) (> (snow ?d) 0))
    :effect (and (decrease (snow ?d)) (increase (snow ?w))))
)
```

Listing 4.2: Lifted QNP Problem for Snow

```
(define (problem snow-01)
  (:domain snow)
  (:objects W_1 - walkway D_1 - driveway)
  (:init (= (snow W_1)  2) (= (snow D_1) 3))
  (:goal (and (= (snow W_1) 0) (= (snow D_1) 0)))
)
```

A class of **generalized QNPs** is then defined as a set $\mathfrak{G} = \{P_1, P_1, ...\}$ of QNP instances $P_i = \langle D_i, I_i \rangle$ that all share the same domain D, but differ in the instances I.

4.1.2 Comparison Graph

The solver in section 4.3 requires the state space to be finite. The reduction from QNPs to FOND formulated by Bonet et al. [3] works by introducing a boolean for every numerical variable and translates the literals $0 = n$ and $0 < n$ into a propositional atom $p_{n=0}$ ($\neg p_{n=0}$). This approach allows a finite abstraction of the state space, but the values of the functions in the states are lost which are needed to evaluate the features in section 4.2.
To keep the state space of the lifted QNP in planning finite and allowing the features to evaluate on real values, we introduce the concept of a comparison graph. A **comparison graph** is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{T})$ where each node $n \in \mathcal{N}$ represents a comparison state. Each comparison state describes a group of states which agree on all boolean predicates and all comparisons in a function set. The function sets are computed based on all preconditions, effects and goal conditions of the domain D. Reducing the comparisons in every comparison state only to the function sets reduces the number of nodes in the graph significantly. Each directed

Algorithm 1 Build Function Sets.

Require: Domain D, Problem P
Ensure: List of function sets Z

```
1  procedure BUILDFUNCTIONSETS(D, P)
2      Z ← [ {f} | f ∈ GROUNDFUNCTIONS(D, P)]
3      Facts ← ∅
4      for all a ∈ GROUNDACTIONS(D, P) do
5          Facts ← Facts ∪ PRECOND(a)
6          Facts ← Facts ∪ EFFECT(a)
7      Facts ← Facts ∪ GOAL(P)
8      for all φ ∈ Facts do
9          (F, C) ← SPLITARGUMENTS(φ)        F = function symbols, C = numeric constants
10         for all f1 ∈ F do
11             for all f2 ∈ F do
12                 z1 ← FINDSETCONTAINING(f1, Z)
13                 z2 ← FINDSETCONTAINING(f2, Z)
14                 if z1 ≠ z2 then
15                     z1 ← z1 ∪ z2                          merge function sets
16                     remove z2 from Z
17         for all f1 ∈ F do
18             for all c1 ∈ C do
19                 z ← FINDSETCONTAINING(f1, Z)
20                 z ← z ∪ {c1}                              add constant
21     return Z
```

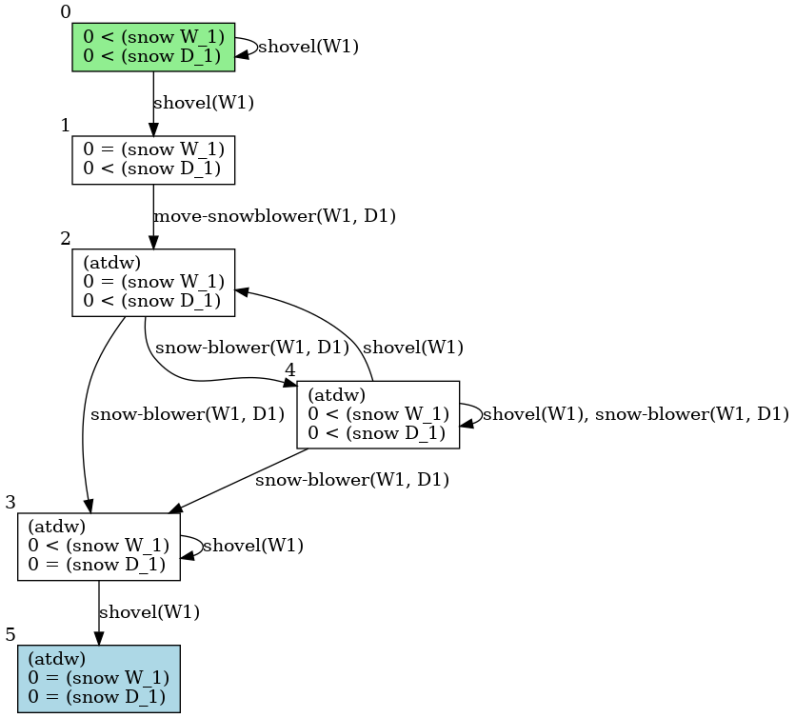
edge $t \in \mathcal{T}$ represents an action that can be applied. An action may have multiple outcomes, leading to different successor states. The graph captures all possible transitions between states based on the actions defined in the QNP. The resulting structure is non-deterministic, but offers a finite representation.

The graph is constructed as follows: *Snow instance with function sets* $\{0, snow(W_1)\}, \{0, snow(D_1)\}$ *as example in grey.*

1. We start with the initial state to build the first comparison state. The function sets are sorted using the equalities from the ground functions to their initial values. The set of predicates of the first comparison state is the same as in the initial state. The initial values are only used to define the ordering of the ground functions in the sets. To reduce the number of states, the initial values are only stored in the comparison graph if they are static. *Initial state* $s_0 = snow(W_1) = 2, snow(D_1) = 3$ *resulting in* $\{0 < snow(W_1)\}$ *and* $\{0 < snow(D_1)\}$.
2. From there, we iteratively expand the graph by applying all applicable actions to each comparison state. The resulting successor node is computed by updating the comparisons and predicates according to the action's effects. *For example, if a variable* $snow(W_1)$ *has the comparisons* $snow(W_1) > 0$ *and is decreased, then, adhering to bounded transitions, this results in two successors. Either* $snow(W_1) > 0$ *or* $snow(W_1) = 0$.
3. A comparison state is a goal state $\mathcal{N}_\mathcal{G} \subset \mathcal{N}$ if it satisfies all goal conditions. The expansion continues until no new states can be generated.

For constructed comparison graph for the example Snow is shown in Figure 4.1.

Figure 4.1: Comparison Graph of the Snow domain, green = initial, blue = goal.



The comparison graphs are used to provide a finite, compact abstraction of the numeric state space by grouping together all ground states that agree on relevant numeric comparisons and

predicate truth values, and by encoding bounded, non-deterministic action outcomes as edges. Unlike the abstraction with FOND, all relevant information about the real values is preserved. The comparisons define restraints with which the graph can be instantiated.
The solver in section 4.3 needs this finite state space to find the actions which lead to the goal and to evaluate the features which are build on grounded functions and evaluate on values.

4.2 Feature Pool

The general policy solver requires a pool of abstract features defined over the QNP state space. We extend the feature language from STRIPS to QNPs, by extending the description logic approach of the DLPlan [4] library by incorporating the ground functions into the vocabulary. To follow the same grammar as outlined by Hofmann et al. [6] for the concepts and roles, we define the universe Δ^s as the set of objects in a planning state s.
While in classical planning domain predicates are either true or false, the lifted representation of QNPs includes real-valued functions that assign a value to objects. A **primitive frame** can either be unary or binary and is build from a lifted function of the problem domain. For example, the function $x(o)$ returns the x-coordinate of an object o and the objects in the domain are $O = \{o_1, o_2, \dots, o_n\}$, then the primitive frame is defined as $(x)^s = \{(o_0, x^s(o_0)), \dots, (o_n, x^s(o_n))\}$. Formally, for each unary function g or binary function h, we create primitive frames:

- Unary frames: $\mathbf{U} = (g)^s := \{(o, v) | o \in \Delta^s,\ v = g^s(o)\}$.
- Binary frames: $\mathbf{B} = (h)^s := \{(o_0, o_1, v) | o_0, o_1 \in \Delta^s,\ v = h^s(o_0, o_1)\}$.

More complex frames can be iteratively built using the following operators, with U, V denoting unary frames, B a binary frame, C a concept, and R a role:

- Concept restriction: $\mathbf{U|C}$ where $(U|C)^s := \{(o, v) \in U^s | o \in C^s\}$.
- Role restriction: $\mathbf{B|R}$ where $(B|R)^s := \{(o_0, o_1, v) \in B^s | (o_0, o_1) \in R^s\}$.
- Distance: $\mathbf{dist(U, V)}$ where $(dist(U, V))^s := \{(o_0, o_1,\ |v_0 - v_1|) | (o_0, v_0) \in U^s,\ (o_1, v_1) \in V^s\}$.

Using these frames, we define features for any frame M (unary or binary) as:

- Maximum: $\mathbf{max(M)}$ where

$$(max(M))^s := \begin{cases} \max\{ v \mid (o, v) \in G^s\}, & \text{if } |M^s| > 0, \\ -\infty, & \text{otherwise.} \end{cases}$$

- Minimum: $\mathbf{min(M)}$ where

$$(min(M))^s := \begin{cases} \min\{ v \mid (o, v) \in G^s\}, & \text{if } |M^s| > 0, \\ \infty, & \text{otherwise.} \end{cases}$$

- Sum: **sum**(**M**) where

$$(sum(M))^s := \sum_{(o,v)\in M^s} v.$$

Analogous to the complexity of a concept or role, the complexity of a frame is the size of its syntax tree. This complexity will be considered in the ASP solver in section 4.3 as the weight $w(f)$ of the feature f. For feature generation we only consider a finite subset of roles, concepts and frames up to complexity bound $c_{\max}$.

For the example domain Snow, the one of the generated features is

n_sum_frame(f_primitive_unary(snow,0)).

The complexity of this feature is 2. It is build iteratively with first the primitive unary snow, describing for every location how much snow is present in the state, and then extracting the sum feature, which describes the sum of all snow present in all locations of a state.

4.3 Selection of Features and Transitions

The policy generation and verification follows the pipeline introduced by Hofmann et al. [6]. It takes a feature pool and state space as input, detects and prunes all dead states unable to reach the goal and defines an ASP solver to find a selection of features and transitions. However, there are modifications necessary in the dead-end detection, ASP encoding and a new termination check which follows the principle of the fairness assumptions.

4.3.1 Dead-end detection

Dead ends are states from which there is no transition to a goal. They are identified because no policy should ever choose an action which may lead to a dead state.
Initially, each state in S is pre-partitioned into three categories alive, dead-end, and goal states. To achieve this, we utilize the dead end detection algorithm described in [6] with the modification that it takes the ground actions A from a lifted QNP instance, the set of comparison states $\mathcal{N}$, and goal states $\mathcal{N}_\mathcal{G}$ from the comparison graph, as input.

Algorithm 2 Dead-End Detection [6].

Require: Ground Actions A, Comparison States N, Goal states N_G

Ensure: Dead-end set $D \subseteq N$

procedure DEADENDDETECTION($M(P)$)
  $D \leftarrow \emptyset$
  repeat
    for all $n \in N \setminus D$ **do**
      for all $a \in A(n)$ **do**
        if $F(a,n) \cap D \neq \emptyset$ **then**
          Remove a from $A(n)$
    for all $n \in N \setminus D$ **do**
      if $\neg\exists$ path $n \xrightarrow{a_1} \ldots \xrightarrow{a_k} n_g$. $a_i \in A(n_i)$, $n_g \in N_G$ **then**
        Add n to D
  until D does not change
  return D

The algorithm $Dead-EndDetection$ works as follows: Goal states are defined to satisfy the goal literals. Iteratively, if applying an action a from a state s leads to a successor state with no path to a goal, action a is excluded from the set of applicable actions in s. States that lose all applicable actions become dead-ends, while states retaining at least one valid action are deemed alive. Thus the algorithm eliminates transitions leading to dead-ends, ensuring that each alive state is connected to another alive or goal state.

4.3.2 Min-Cost SAT Solver

For the solver, the input are the domain D, the problem instances P_i, and the comparison graphs $\mathcal{G}_i = (\mathcal{N}_i, \mathcal{T}_i)$ with the generated feature pool F. The output is a set of good transitions corresponding to a policy that reaches the goal, along with a set of features that distinguish good from non-good transitions. The policy can be generated as described in section 4.4.
First, we formulate the problem instances and the features as well as the evaluation of the features as facts encoded in ASP. Then for the rules and conditions of the transition, the features and the termination utilizing the fairness assumption the ASP solver finds a satisfiable assignment of selected features and good transitions, which are forwarded in the pipeline.

ASP Encoding of the Problems

From the comparison graph and dead-end information, we create an ASP encoding to represent the states, transitions, features and feature evaluations. This means for comparison states S in $\mathcal{N}_\mathcal{I}$ of problem instance I we create facts **state(I, S)** and additionally **alive(I, S)** or **goal(I,S)**, if the state is alive or a goal respectively. For each transition $(s, a, s') \in \mathcal{T}_\mathcal{I}$ we

create facts **trans(I, S, A, S')** where S is the source state, A is the action, and S' is the target state. The features f and their respective complexity c are encoded as **feature(F)** and **feature_complexity(F, C)**.
Next, the comparison graph is instantiated with values and serves as a finite representation of the lifted QNP for evaluating the features. Each comparison state is instantiated by assigning values that satisfy the comparisons and predicates of that state. Features are then evaluated on these instantiated states. The resulting facts are **eval(I, S, F, D)** with the problem instance I, the comparison state S, feature F and the evaluated feature value D.

The following code shows part of the ASP encoding of the Snow instance, declaring a boolean feature which describes if the snowblower is at the driveway and two numericals computing the maximum and overall amount of snow in a state. Exemplarily, the state s^2 of the instance 0 is displayed, complete with the evaluations of the features. Additionally, we declare "snowblower(W_1,D_1)" as a grounded action of snowblower(x,y) and the transition of that action from state s^2 to s^3.

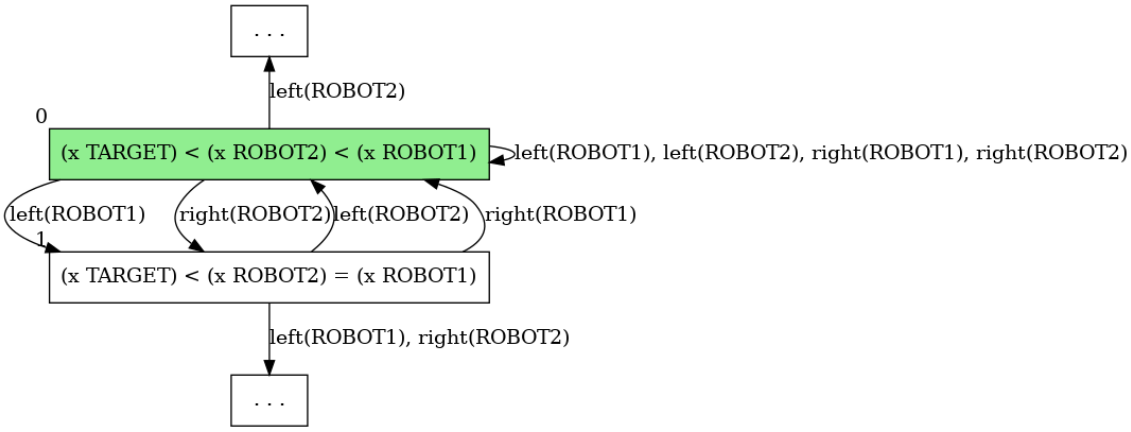
Listing 4.3: Snippet 1 of the ASP encoding of the Snow instance.

```
1       feature("b_nullary(atdw)").
2       feature_complexity("b_nullary(atdw)", 1).
3       feature("n_maximum(f_primitive_unary(snow,0))").
4       feature_complexity("n_maximum(f_primitive_unary(snow,0))", 2).
5       feature("n_sum_frame(f_primitive_unary(snow,0))").
6       feature_complexity("n_sum_frame(f_primitive_unary(snow,0))", 2).
7       [...]
8       state(0, 2).
9       alive(0, 2).
10      eval(0, 2, "b_nullary(atdw)", 1).
11      eval(0, 2, "n_minimum(f_primitive_unary(snow,0))", 0).
12      eval(0, 2, "n_sum_frame(f_primitive_unary(snow,0))", 100).
13      aname("snow-blower(W_1,D_1)", "snow-blower").
14      trans(0, 2, "snow-blower(W_1,D_1)", 3).
15
```

Similar to the eval(I, S, F, D), we encode the change of the features along a transition $(s, a, s') \in \mathcal{T}$ with the term **trans_delta(I, S, A, S' F, D)**. With the change D encoding an increase of a feature with 1, no change with 0, and a decrease with -1.
Comparison graphs offer a very compact way to abstract from the numerical state space. However, for the evaluation of the change D in a feature along a transition, there is an occurance where the graph is too general and may lead to some features not being evaluated correctly. This means that here for each trans_delta(I, S, A, S', F, D), the program checks whether or not

the values violate the effects of the actions and if so evaluate on a corrected state.
The following example illustrates this issue. In the one dimensional go domain the objects include multiple robots and multiple locations. The robots are able to move left and right, and the goal is for them to reach their respective target locations. The problem instance here contains two robots ROBOT1 and ROBOT2 which both need to reach the target TARGET.

Figure 4.2: Go domain with two robots which have the same target, only two nodes displayed.



Exemplarily for the evaluation problem, we will consider the transitions from state s_0 to s_1, namely $(left(ROBOT1))$ and $(right(ROBOT2))$, together with the feature f = ”n_maximum(f_primitive_unary(x,0))”. The states are instantiated with:

$$\{x(ROBOT1) = 2.0, x(ROBOT2) = 1.0, x(TARGET) = 0.0\}$$
$$\text{for } s_0 \text{ which evaluates to } (f)^{s_0} = 2.0, \text{and}$$
$$\{x(ROBOT1) = 1.0, x(ROBOT2) = 1.0, x(TARGET) = 0.0\}$$
$$\text{for } s_1 \text{ which evaluates to } (f)^{s_1} = 1.0.$$

These values satisfy the comparisons and represent the feature change along the transition $(s_0, left(ROBOT1), s_1)$

$$trans\_delta(I, s_0, left(ROBOT1), s_1, n\_maximum(f\_primitive\_unary(x, 0)), -1).$$

However, for the action right(ROBOT2) with the effect inc(ROBOT2) the value of $x(ROBOT1)$ remains at 2.0 and $x(ROBOT2)$ increases. The additional check of the action effects catches the violation and produces a corrected state

$$\{x(ROBOT1) = 2.0, x(ROBOT2) = 2.0, x(TARGET) = 0.0\}$$

called s_1' which then evaluates the feature to $(f)^{s_1'} = 2.0$ and the feature change to

$$trans\_delta(I, s_0, right(ROBOT2), s_1, n\_maximum(f\_primitive\_unary(x, 0)), 0)$$

For the snow domain, the transition evaluation continues as shown below.

Listing 4.4: Snippet 2 of the ASP encoding of the Snow instance.

```
1       trans_delta(0, 2, "snow-blower(W_1,D_1)", 3, "b_nullary(atdw)", 0).
2       trans_delta(0, 2, "snow-blower(W_1,D_1)", 3, "b_nullary(atdw_G)", 0).
3       ...
4       trans_delta(0, 2, "snow-blower(W_1,D_1)", 3, "n_minimum(
         f_primitive_unary(snow,0))", 1).
5       trans_delta(0, 2, "snow-blower(W_1,D_1)", 3, "n_maximum(
         f_primitive_unary(snow,0))", 0).
6       trans_delta(0, 2, "snow-blower(W_1,D_1)", 3, "n_sum_frame(
         f_primitive_unary(snow,0))", 1).
7       [...]
8
```

Good Transition Encoding

As initial framework, we use the ASP encoding from Hofmann et al.[6], but we modify the termination and good transitions to formulate a complete solver for the extended QNPs. The solver is explained step by step, and finally summarized in it's essential points. The termination condition is explained seperately.
The solver finds a satisfiable assignment, which features need to be **selected()** and which transitions are choosen to lead to the goal states **good_trans()**.

Listing 4.5: Good Transition and Feature Encoding (ASP) Part 1.

```
1     % There must be a good transition for every alive non-goal state that ends
          in an alive state.
2     :- alive(I, S), not good_trans(I, S, _, _), not goal(I, S).
3     % If a state is not alive, then it has no good transitions
4     :- good_trans(I, _, _, S), not alive(I, S).
5     % Select at least one feature
6     { selected(F) } :- feature(F).
7
```

Furthermore, the selected features should be able to tell apart all goal from non-goal states. The binary encoding is relevant, because in the general policy we are reduced to binary feature

conditions. This means that for state evaluations, we are only able to distinguish if a feature f is True (f = 0) or false (f != 0).

Listing 4.6: Good Transition and Feature Encoding (ASP) Part 2.

```
1       % Translate feature evaluations in states into in binary encoding
2       bool_eval(I, S, F, 1) :- eval(I, S, F, V), V > 0.
3       bool_eval(I, S, F, 0) :- eval(I, S, F, V), V = 0.
4       % Bool_dist describes if a selected feature is able to differentiate two
          states
5       bool_dist(I1, S1, I2, S2) :- state(I1, S1), state(I2, S2), bool_eval(I1,
          S1, F, V1), bool_eval(I2, S2, F, V2), selected(F), V1 != V2.
6       % If so, then the features tell apart goal from non goal-states.
7       :- state(I1, S1), state(I2, S2), not bool_dist(I1, S1, I2, S2), goal(I1,
         S1), not goal(I2, S2).
8
```

Additionally, it is forbidden to have two alive states that look the same (not bool_dist), where there is a good transition from the first state, and there exists a comparable transition from the second state that is not marked “good,” unless those two transitions are known to be distinguishable.

Listing 4.7: Good Transition and Feature Encoding (ASP) Part 3.

```
1     % Only Transitions that apply a different change to a feature F are
        considered different
2     trans_diff(F, I1, S11, A1, S12, I2, S21, A2, S22) :- trans_delta(I1, S11,
        A1, S12, F, D1), trans_delta(I2, S21, A2, S22, F, D2), D1 != D2.
3     % Trans_diff is called distinguished if the feature is selected
4     distinguished(I1, S11, A1, S12, I2, S21, A2, S22) :- trans_diff(F, I1, S11,
         A1, S12, I2, S21, A2, S22), selected(F).
5     % then two non-differentiable alive states with two non distinuishable
        transitions cant be one a good transition and the other not
6     :- alive(I1, S11), alive(I2, S21), not bool_dist(I1, S11, I2, S21),
        good_trans(I1, S11, A1, S12), trans(I2, S21, A2, S22), not good_trans(
        I2, S21, A2, S22), not distinguished(I1, S11, A1, S12, I2, S21, A2, S22).

7
```

Finally, a good transition can be picked for each alive state, which is an outcome of a safe action. This means, that for all alive states we have at least one good transition, which will traverse the alive states and to the goal.

Listing 4.8: Good Transition Encoding (ASP) Part 4.

```
% An action is safe, if all transitons lead to alive states
safe_action(I, S1, A) :- state(I, S1), trans(I, S1, A, _), alive(I, S2) :
   trans(I, S1, A, S2).

% Choose at least one good transition from the safe actions.
1 { good_trans(I, S1, A, S2) : trans(I, S1, A, S2), safe_action(I, S1, A) }
    :- alive(I, S1), not goal(I, S1).

% A dead state is considered critical, if it can be reached through a
   transition from an alive state
crit_state(I, S2) :- alive(I, S1), trans(I, S1, _, S2), not alive(I, S2).
% The selected features can tell alive and critical states apart
:- alive(I1, S1), crit_state(I2, S2), not bool_dist(I1, S1, I2, S2).

```

To summarize, every alive state has at least one good transition, which will always lead to another alive state or goal, and the selected features $F' \subseteq F$ are able to distinguish goal from non-goal states, alive and critical dead states, and good from not good transitions. The next section ensures that the good transitions don't circle endlessly, and, if traversed, will eventually reach a goal state.

Termination Encoding

Fairness in QNPs requires that for all features that are decreased in a trajectory, all cycles in which this feature is increased must contain a terminating state. To encode this condition, we modify the ASP by Rodriguez et al. [8] and define the terminating conditions for a state as follows.
A terminating state is:

- any goal state
- any state which is fair and has a good tranition to a terminating state
- any state that, if unfair, has good transitions only to terminating states

We require that every alive state has to be terminating, meaning it has to fullfill these conditions.

Listing 4.9: Termination Encoding (ASP) similar to [8]. The blue clause defines the fairness definition of QNPs.

```
% All states are connected to themselves and all reachable states via good transitions
connected(I, S, S) :- state(I, S).
connected(I, S, T) :- connected(I, S, X), good_trans(I, X, _, T).

% There is no path from S to T, OR all paths contain a terminating state
blocked_paths(I, S, T) :- state(I, S), state(I, T), not  connected(I, S, T).
blocked_paths(I, S, T) :- connected(I, S, T), terminates(I, S).
blocked_paths(I, S, T) :- connected(I, S, T), terminates(I, T).
blocked_paths(I, S, T) :- S != T, connected(I, S, T), blocked_paths(I, X, T): good_trans(I, S, _, X), S != X.

% All cycles containing S and T are blocked (meaning they contain terminating states)
blocked_cycles(I, S, T) :- blocked_paths(I, S, T).
blocked_cycles(I, S, T) :- blocked_paths(I, T, S).

% A state is fair: If there is a feature f that is decreased, then all cycles in which F is increased contain a terminating state
fair(I, S1) :- good_trans_delta(I, S1, A1, S2, F, -1), blocked_cycles(I, S1, S3) : good_trans(I, S3, A2, S4), trans_delta(I, S3, A2, S4, F, 1).

% Terminating states include: goal states; fair states from where there is a good transition to a terminating state; and non-fair states where all transitions lead to terminating states
terminates(I, S) :- goal(I, S).
terminates(I, S) :- fair(I, S), good_trans(I, S, _, T), terminates(I, T).
terminates(I, S) :- not fair(I, S), good_trans(I, S, _, _), terminates(I, T) : good_trans(I, S, _, T).
% Every alive state terminates
:- alive(I, S), not terminates(I, S).

```

Cost reduction

To minimize the number of features selected, we use the same cost reduction as in [6]. This means that we minimalize the sum of the feature complexities by enforcing the constraint $minimize\{C@3, F : selected(F), feature\_complexity(F, C)\}$.

4.4 General Policy

The output of the solver in section 4.3 is a set of features $F' \subseteq F$ and a set of good transitions which lead from any alive state to the goal. The general policy π defined over these features that solves all instances in the generalized QNP $\mathfrak{G}$ is extracted as follows.

4.4.1 Rule construction

For every good_trans(I, S, A, S') a policy rule $C \mapsto E$ is generatey by considering all selected features.

1. The set of **conditions C** is build by mapping the boolean feature evaluations bool_eval(I, S, F, V)
   - for boolean features to V= 0 $\rightarrow \neg b$, V= 1 $\rightarrow b$ and
   - for numerical features to V= 0 $\rightarrow n = 0$, V= 1 $\rightarrow n \neq 0$.
2. The set of **effects E** is build by mapping the trans_delta(I, S, A, S' F, D)
   - for boolean features to D= $-1 \rightarrow \neg b$, D= 1 $\rightarrow b$ and
   - for numerical features to D= $-1 \rightarrow\downarrow n$, D= 1 $\rightarrow\uparrow n$.

If a feature is not affected by the transition (D = 0), then it is not mentioned in the effects. This means that a rule with empty effects has to choose an action which does not change any of the features.
The resulting policy is simplified by removing any rules which are a subset of another rule, and additionally replacing every two rules, which have the same effects and conditions and differ in only one of the condition values, with a new rule which drops that condition.
The final policy consisting of the set of rules $C \mapsto E$ is tested on additional instances, and if it fails, the pipeline is restarted with the additional instance.

For example, we review the general policy found for the snow domain with a single feature $S_{snow} \equiv$ *n_sum_frame(f_primitive_unary(snow,0))* which denotes the sum of all snow variables:

Figure 4.3: Policy for Snow

$$\{S_{snow} > 0\} \Rightarrow \{\}$$
$$\{S_{snow} > 0\} \Rightarrow \{\downarrow S_{snow}\}$$

The first rule states that if possible, then we apply an action that does not change the total amount of snow. The only action that satisfies this is to move the snowblower. The second rule states that if there is still snow, we should apply an action that decreases the total amount of snow. This is achieved by either shoveling or using the snowblower.

5 Evaluation

To evaluate our approach, we present experimental results on several domains with varying complexity. There are existing domains for QNPs, but generalizing to varying problem sizes is a novel task. We evaluate our methods on a domain that has been previously used in the literature for qualitative numerical planning as well as several domains that are new but are inspired by benchmark domains from the literature of classical and numerical domains.

5.1 Domains

A important new aspect of the possible domains we cover, is that we are able to work with continuous space. This is why many of the following domains have been tested in the one dimensional as well as in the two dimensional case.
The snow problem is used to showcase that the proposed approach can solve simple QNPs easily. The GoTo domain is the simplest domain in continuous space and due to its simplesness allows us to show how many of the methods work. Finally the Sailing and Delivery domain represent more complex solutions. The challenge sailing poses, is that not only do we generalize over the boats, but also multiple drowning people. The continuous delivery requires the solution to consider complex sequence of actions, as we do have the constraint, that a package has to be delivered before the gripper can pick a new one up.

Snow

The snow domain was introduced in Section 4.1. The objects are the walkways and the driveway, the domain variables include one function which returns the snow on any location and a boolean which declares if the snowblower is in the driveway. The actions are to *shovel*(w) and thus reducing the snow in any of the walkways, to *move*() the snowblower to the driveway for which the precondition is that on of the walkways has to be free of snow, and using the *snowblower*() which requires that the snowblower is in the driveway and using the snowblower may increase the amount of snow on one of the walkways. This domain was generalized by introducing multiple walkways and different amounts of initial snow at the locations.

GoTo

This domain has been defined in section 4.3.2. In the one dimensional domain the robots are able to move left or right and in the two dimensional domain additionally up or down. The goal for the robots is to reach the target location. We generalize over the number of robots and

arbitrary robot and target coordinates.
For the two dimensional setting, the target is set to the coordinates (0,0) for all instances, because otherwise the ASP code was too large for the solver to handle.

Sailing

In the sailing domain multiple drowning people are lost at sea and need to be saved. Multiple sailing boats are available to reach them. The available functions are only the coordinates of the boats and the people to be rescued, while predicates tell if a person has been saved. The actions include for the boats to sail in all directions and save a drowning person if they have reached them.

Delivery

In the delivery domain, multiple packages need to be delivered to a target location by a gripper, which can move into all directions, pick up, and drop packages. The variables include functions for the coordinates of all objects and the target, and booleans indicating whether the gripper is free or which package it is holding.

5.2 Results

The following table summarizes the results for the domains defined above. We will closely inspect the solutions of the go domain, one dimensional delivery and sailing. Policies, comparison graphs and additional definitions are included in the appendix.

Table 5.1: Evaluation results with $|P|$ is the number of problems, $|S|$ is the number of solved problems, $|T|$ is the number of problems used in training, $|O|_T$ and $|O|_P$ denote the maximum number of objects in the trainig instances and overall respectively, the two time colums represent the time taken to solve the SAT formulation and the total runtime (wall) of the program, mem/MB is the memory needed in megabytes, |F| is the number of features generated, $k*$ is the maximum feature complexity needed and c_Φ is the total cost of the features in the final solution.

$\mathcal{Q}$	$\lvert P\rvert$	$\lvert S\rvert$	$\lvert T\rvert$	$\lvert O\rvert_T$	$\lvert O\rvert_P$	t_{solve}/s	t_{wall}/s	mem/MB	$\lvert\mathcal{F}\rvert$	k^*	c_Φ
qnp_snow	10	10	1	2	7	<0.1	11	46	8	2	2
go_to_1D	10	10	3	4	11	120	105	12 388	27	6	8
go_to_2D	5	5	2	3	5	91 940	3016	59 166	35	4	8
sailing	5	**4**$^{\mathbf{C}}$	3	7	7	5	97	1358	138	7	14
delivery1D	10	10	4	4	8	210	744	9341	68	6	21

The results to running the experiments with a memory limit of 248 GB and a maximal feature complexity cmax = 15 are shown in table 5.1. The policies were generated with problems

containing fewer objects, but generalized well to additional objects. Only with sailing did the policy fail on the more complex instances, and failed to generate a better policy due to the cmax constraint.

GoTo

The GoTo domain is the simplest domain possible which defines continuous space in one or two dimensions. The policies are therefore expectantly simple. For the one dimensional variant, the only feature selected is

$SumDis_{\text{x}} \equiv$ n_sum_frame(f_distance(f_restrict_unary(f_primitive_unary(x,0), c_primitive(is_robot,0)), f_restrict_unary(f_primitive_unary(x,0), c_primitive(is_target,0))))

which describes the sum of all distances along the x-Axis of all robots to the target location.

Figure 5.1: Policy for One dimensional GoTo

$$\{SumDis_x > 0\} \Rightarrow \{\downarrow SumDis_x\}$$

Sailing

The failing policy in the sailing domain has 4 features,

$$U \equiv \text{n\_count(c\_diff(c\_primitive(saved\_G,0),c\_primitive(saved,0)))}$$

is the number of not yet saved people, and

$MD_{\text{StoS}} \equiv$ n_minimum(f_distance(f_restrict_unary(f_primitive_unary(x,0), c_not(c_primitive(saved,0))), f_restrict_unary(f_primitive_unary(x,0), c_not(c_primitive(saved,0)))))

is the distance of the two closest not saved objects.

Figure 5.2: Policy for Two dimensional GoTo

$$\begin{aligned}
\{U > 0 \wedge MD_{StoT} = 0\} \Rightarrow &\quad \{\uparrow MD_{StoS}\} \\
\{U > 0 \wedge MD_{StoT} = 0\} \Rightarrow &\quad \{\downarrow U\} \\
\{U > 0 \wedge MD_{StoT} = 0\} \Rightarrow &\quad \{\downarrow U \wedge \uparrow MD_{StoS}\} \\
\{U > 0 \wedge MD_{StoT} > 0\} \Rightarrow &\quad \{\downarrow MD_{StoS}\}
\end{aligned}$$

The policy fails as soon as a domain has more people to save, because it is unable to distinctivly pick out the distances from the boat to the people. The needed features are in sum too costly for the cmax restraint to generate and the solver fails.

Delivery

For the one dimensional domain, the generated policy has four features

$$F \equiv \text{b\_nullary(free)}$$

tells whether the gripper is free,

$$MD \equiv \text{n\_maximum(f\_distance(f\_primitive\_unary(x,0), f\_primitive\_unary(x,0)))}$$

denotes the maximum distance between two objects in a state, whereas

$$MD_{\text{PtoT}} \equiv \text{n\_maximum(f\_distance(f\_restrict\_unary(f\_primitive\_unary(x,0), c\_primitive(is\_p,0)), f\_restrict\_unary(f\_primitive\_unary(x,0), c\_primitive(is\_t,0))))}$$

is the maximum distance specified between all packages and the target, and finally

$$D_{\text{GtoT}} \equiv \text{n\_minimum(f\_distance(f\_restrict\_unary(f\_primitive\_unary(x,0), c\_primitive(is\_g,0)), f\_restrict\_unary(f\_primitive\_unary(x,0), c\_primitive(is\_t,0))))}$$

is the distance of the gripper to the target.

Figure 5.3: Policy for One dimensional Delivery

$$\begin{aligned}
\{F \wedge MD > 0 \wedge MD_{PtoT} > 0\} &\Rightarrow \{\uparrow D_{GtoT}\} \\
\{F \wedge MD > 0 \wedge MD_{PtoT} > 0 \wedge D_{GtoT} > 0\} &\Rightarrow \{\neg F\} \\
\{MD = 0 \wedge MD_{PtoT} = 0 \wedge D_{GtoT} = 0 \wedge \neg F\} &\Rightarrow \{F\} \\
\{MD > 0 \wedge MD_{PtoT} > 0 \wedge D_{GtoT} = 0 \wedge \neg F\} &\Rightarrow \{F\} \\
\{MD > 0 \wedge MD_{PtoT} > 0 \wedge D_{GtoT} = 0 \wedge \neg F\} &\Rightarrow \{\downarrow MD_{PtoT} \wedge F \wedge \downarrow MD\} \\
\{MD > 0 \wedge MD_{PtoT} > 0 \wedge D_{GtoT} > 0\} &\Rightarrow \{\downarrow D_{GtoT} \wedge \downarrow MD\} \\
\{MD > 0 \wedge MD_{PtoT} > 0 \wedge D_{GtoT} > 0 \wedge \neg F\} &\Rightarrow \{\downarrow D_{GtoT}\} \\
\{MD > 0 \wedge D_{GtoT} > 0 \wedge \neg F\} &\Rightarrow \{\downarrow D_{GtoT} \wedge \downarrow MD\}
\end{aligned}$$

The policy is complex, but can be broken down into sections. Rule 1 and 2 define that if the gripper is free, we move away from the target and then try to pick up a package. If the gripper is at the target and holding a package, then rule 3, 4 and 5 say to drop it. Whenever there are still packages, the gripper is not at the target, too far from any object to pick it up, then the rule 6 tries to decresses the distance of the furthest two objects. Rules 7 and 8 decrease the distance of the gripper to the target, if it holds a package.

6 Conclusion and Future Work

This thesis has presented an approach for learning general policies for QNPs that scales from single-instance reasoning to entire classes of problems sharing a common lifted domain description. We extended the original QNP formalism with additional numeric comparisons and assignment effects. The learning approach is based on comparison graphs, which provide a finite abstraction of the infinite state space, while preserving all relevant information. As they adhere to ϵ- and ω-bounded transitions, the transitions in the comparison graphs will not over- or undershooting numeric thresholds.
Building on this representation, we expanded the DLPlan feature-generation framework with numeric frames that incorporate grounded functions, yielding new expressive numerical features such as restrictions, distances, maxima, minima, and aggregation over the functions of the continuous domains. The revised ASP encoding is fitted to handle QNPs as well and includes a new termination check which follows the fairness assumptions.
Finally, evaluation across four domains, including continuous Delivery in one- and two-dimensional continuous space, demonstrated the effectiveness of the proposed approach. The learned policies solved all benchmark instances.
Overall, this work shows that generalized planning and qualitative numerical reasoning can be successfully combined, enabling automatic derivation of reusable policies for domains that feature both discrete structure and continuous resources. By unifying advances in lifted representations, description-logic-based feature synthesis, and ASP-based policy generation, we provide a framework that extends the boundaries of what generalized planning can address.

Future Work

While the proposed pipeline succeeds on the provided benchmark set, several challenges remain open for exploration:

- More features: Building additional features might improve policy size and helps solving more complex domains.
- Nullary frames: Extending frames to allow direct comparison of function outputs with constants may reduce complexity.
- Redundancy elimination: Some rules are still redundant and at a higher complexity make policies hard to read.
- Integration with motion planning: Coupling the high-level generalized policies with low-level continuous motion planners

By pursuing these directions, future research can make generalized qualitative numerical planning an even more powerful tool for tackling large families of continuous problems.

A Appendix

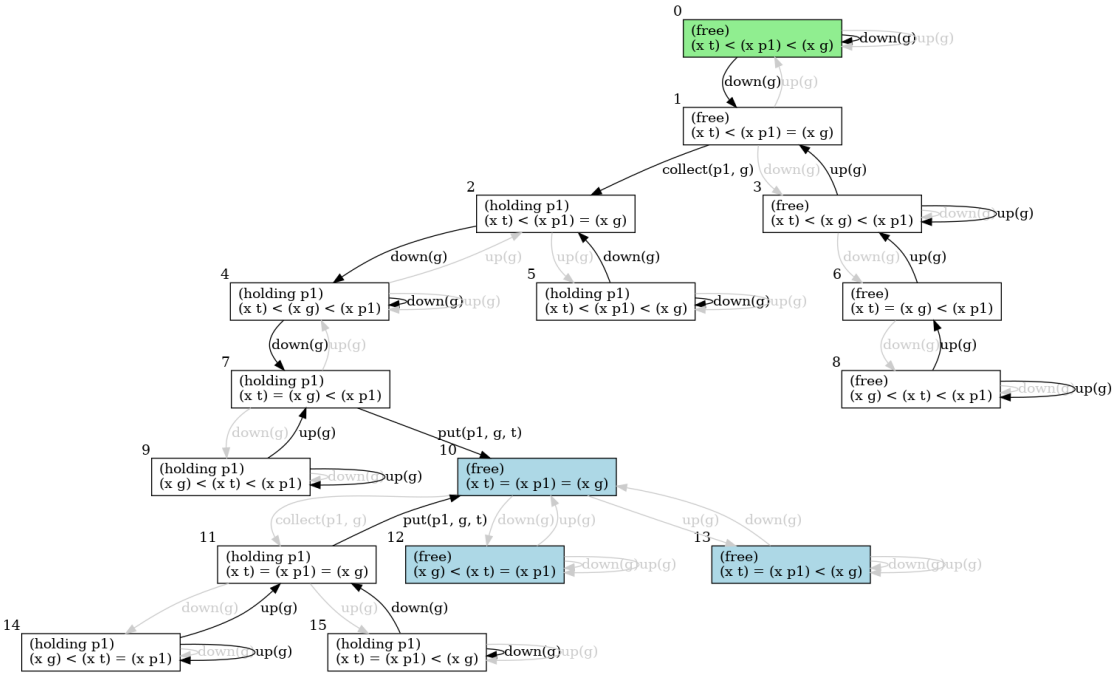
Listing A.1: Lifted QNP Domain for Delivery 2D

```
(define (domain delivery)
    (:requirements :strips :numeric-fluents :quantified-preconditions :typing)
    (:types package gripper target - locationable)
    (:predicates (holding ?p - package) (free) (is_p ?p - package) (is_g ?g - gripper) (is_t ?t - target))
    (:functions (x ?o - locationable) (y ?o - locationable))
    (:action up
        :parameters (?g - gripper)
        :precondition (and )
        :effect (and (increase (y ?g) 1)))
    (:action down
        :parameters (?g - gripper)
        :precondition (and )
        :effect (and (decrease (y ?g) 1)))
    (:action right
        :parameters (?g - gripper)
        :precondition (and )
        :effect (increase (x ?g) 1))
    (:action left
        :parameters (?g - gripper)
        :precondition (and )
        :effect (decrease (x ?g) 1))
    (:action collect
        :parameters (?p - package ?g - gripper)
        :precondition (and (= (x ?p) (x ?g)) (= (y ?p) (y ?g)) (free))
        :effect (and (holding ?p) (not (free))))
    (:action put
        :parameters (?p - package ?g - gripper ?t - target) ; find the pkg at the target and put it down there
        :precondition (and (holding ?p) (= (x ?t) (x ?g)) (= (y ?t) (y ?g)))
        :effect (and (not (holding ?p)) (free) (assign (x ?p) (x ?t)) (assign (y ?p) (y ?t))))
)
```

Listing A.2: Lifted QNP Problem with 1 Package for Delivery 2D

```
(define (problem p1)
    (:domain delivery)
    (:objects p1 - package g - gripper t - target)
    (:init (= (x p1) 3) (= (y p1) 1)
           (= (x t) 0) (= (y t) 0)
           (= (x g) 5) (= (y g) 2)
           (free) (is_p p1) (is_g g) (is_t t))
    (:goal (and (= (x p1) (x t)) (= (y p1) (y t))
           (free) (not (holding p1)))))
```

Figure A.1: Comparison Graph of the Delivery 1D domain, green = initial, blue = goal, black edges = good_trans.



List of Acronyms

ASP Answer Set Program

FOND Fully Observable Non-Deterministic

QNP Qualitative Numeric Planning

List of Figures

List of Tables

List of Algorithms

List of References

[1] M. Aichmüller and H. Geffner, "Sketch decompositions for classical planning via deep reinforcement learning," in *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-25)*, IJCAI Organization, 2025, pp. 8438–8446.

[2] B. Bonet, G. Francès, and H. Geffner, *Learning features and abstract actions for computing generalized plans*, Nov. 17, 2018. DOI: 10.48550/arXiv.1811.07231 arXiv: 1811.07231[cs]. [Online]. Available: http://arxiv.org/abs/1811.07231

[3] B. Bonet and H. Geffner, "Qualitative numeric planning: Reductions and complexity," *Journal of Artificial Intelligence Research*, vol. 69, pp. 923–961, 2020.

[4] D. Drexler, G. Francès, and J. Seipp, *DLPlan*, 2022. DOI: 10.5281/zenodo.5826139 [Online]. Available: https://doi.org/10.5281/zenodo.5826139

[5] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[6] T. Hofmann and H. Geffner, *Learning generalized policies for fully observable non-deterministic planning domains*, May 13, 2024. DOI: 10.48550/arXiv.2404.02499 arXiv: 2404.02499[cs]. [Online]. Available: http://arxiv.org/abs/2404.02499

[7] L. Illanes and S. A. McIlraith, "Generalized planning via abstraction: Arbitrary numbers of objects," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, pp. 7610–7618, 2019. DOI: 10.1609/aaai.v33i01.33017610

[8] I. D. Rodriguez, B. Bonet, S. Sardina, and H. Geffner, "Flexible FOND planning with explicit fairness assumptions," *Journal of Artificial Intelligence Research*, vol. 74, Jun. 23, 2022, ISSN: 1076-9757. DOI: 10.1613/jair.1.13599 arXiv: 2103.08391[cs]. [Online]. Available: http://arxiv.org/abs/2103.08391

[9] S. Srivastava, N. Immerman, and S. Zilberstein, "A new representation and associated algorithms for generalized planning," *Artificial Intelligence*, vol. 175, no. 2, pp. 615–647, 2011, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2010.10.006 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370210001761

[10] S. Srivastava, S. Zilberstein, N. Immerman, and H. Geffner, "Qualitative numeric planning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 25, 2011, pp. 1010–1016.

[11] S. Ståhlberg, B. Bonet, and H. Geffner, "Learning generalized policies without supervision using gnns," in *Proceedings of the Nineteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2022)*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2022, pp. 474–483. DOI: 10.24963/kr.2022/49